



A T M E
College of Engineering



Module 2

Learning Sets of Rules



A T M E
College of Engineering



Contents

Learning Sets of Rules: Sequential Covering Algorithms, Learning Rule Sets: Example-Based Methods, Learning First-Order Rules, FOIL: A First-Order Inductive Learner.

Analytical Learning: Perfect Domain Theories: Explanation-Based Learning, Explanation-Based Learning of Search Control Knowledge, Inductive-Analytical Approaches to Learning



Learning Sets of Rules

Introduction

- One of the most expressive and human readable representations for learned hypotheses is **sets of if-then rules**
- In many cases it is useful to learn the target function represented as a set of **if-then rules** that jointly define the function.
 - one way to learn sets of rules is to first learn a decision tree, then translate the tree into an equivalent set of rule
 - A second method is to use a genetic algorithm that encodes each rule set as a bit string and uses genetic search operators to explore this hypothesis space



A T M E
College of Engineering



- A **variety of algorithms** that directly learn rule sets and that differ from these algorithms in two key respects
 - **First**, **they are designed to learn sets of first-order rules** that contain variables. This is significant because first-order rules are much more expressive than propositional rules.
 - **Second**, the algorithms discussed here **use sequential covering algorithms** that learn one rule at a time to incrementally grow the final set of rules.



SEQUENTIAL COVERING ALGORITHMS

- Here we consider a **family of algorithms for learning rule sets** based on the **strategy of learning one rule**, removing the data it covers, then iterating this process. Such algorithms are called **sequential covering algorithms**
- **To elaborate**, imagine we have a **subroutine LEARN-ONE-RULE** that **accepts a set of positive and negative training examples** as input, then **outputs a single rule** that covers many of the positive examples and few of the negative examples.
- We require that **this output rule** have **high accuracy**, but **not necessarily high coverage**. By high accuracy, we mean the predictions it makes should be correct.
- By accepting low coverage, we mean it need not make predictions for every training example.



- Given this **LEARN-ONE-RULE** subroutine for learning a single rule, one obvious approach to learning a set of rules is to invoke **LEARN-ONE-RULE** on all the available training examples, remove any positive examples covered by the rule it learns, then invoke it again to learn a **second rule** based on the remaining training examples.
- This procedure can be iterated as many times as desired to learn a disjunctive set of rules that together cover any desired fraction of the positive examples.
- This is called a **sequential covering algorithm** because it sequentially learns a set of rules that together cover the full set of positive examples.
- The **final set of rules** can then be sorted so that more accurate rules will be considered first when a new instance must be classified
- This sequential covering algorithm is one of the most widespread approaches to learning disjunctive sets of rules

SEQUENTIAL-COVERING(*Target_attribute*, *Attributes*, *Examples*, *Threshold*)

- *Learned_rules* $\leftarrow \{\}$
 - *Rule* $\leftarrow$ LEARN-ONE-RULE(*Target_attribute*, *Attributes*, *Examples*)
 - while PERFORMANCE(*Rule*, *Examples*) > *Threshold*, do
 - *Learned_rules* $\leftarrow$ *Learned_rules* + *Rule*
 - *Examples* $\leftarrow$ *Examples* - {examples correctly classified by *Rule*}
 - *Rule* $\leftarrow$ LEARN-ONE-RULE(*Target_attribute*, *Attributes*, *Examples*)
 - *Learned_rules* $\leftarrow$ sort *Learned_rules* accord to PERFORMANCE over *Examples*
 - return *Learned_rules*
-

TABLE 10.1

The sequential covering algorithm for learning a disjunctive set of rules. LEARN-ONE-RULE must return a single rule that covers at least some of the *Examples*. PERFORMANCE is a user-provided subroutine to evaluate rule quality. This covering algorithm learns rules until it can no longer learn a rule whose performance is above the given *Threshold*.



A T M E
College of Engineering



- It reduces the problem of learning a disjunctive set of rules to a sequence of simpler problems, each requiring that a single conjunctive rule be learned
- **Because it performs a greedy search**, **formulating a sequence of rules without backtracking**, it is not guaranteed to find the smallest or best set of rules that cover the training examples.



1. General to Specific Beam Search

- One effective approach to implementing LEARN-ONE-RULE is to organize the hypothesis space search in the same general fashion as the ID3 algorithm
- The search begins by considering the most general rule precondition possible (the empty test that matches every instance), then greedily adding the attribute test that most improves rule performance measured over the training examples.
- Once this test has been added, the process is repeated by greedily adding a second attribute test, and so on
- Like ID3, this process grows the hypothesis by greedily adding new attribute tests until the hypothesis reaches an acceptable level of performance.
- Unlike ID3, this implementation of LEARN-ONE RULE follows only a single descendant at each search step- the attribute-value pair yielding the best performance



- This approach to implementing LEARN-ONE-RULE performs a general-to specific search through the space of possible rules in search of a rule with high accuracy, though perhaps incomplete coverage of the data.
- The general-to-specific search suggested above for the LEARN-ONE-RULE algorithm is a greedy depth-first search with no backtracking
- As with any greedy search, there is a danger that a suboptimal choice will be made at any step.
- To reduce this risk, we can extend the algorithm to perform a beam search; that is, a search in which the algorithm maintains a list of the k best candidates at each step, rather than a single best candidate.
- Beam search keeps track of the most promising alternatives to the current top-rated hypothesis, so that all of their successors can be considered at each search step.
- This general to specific beam search algorithm is used by the CN2 program described by Clark and Niblett

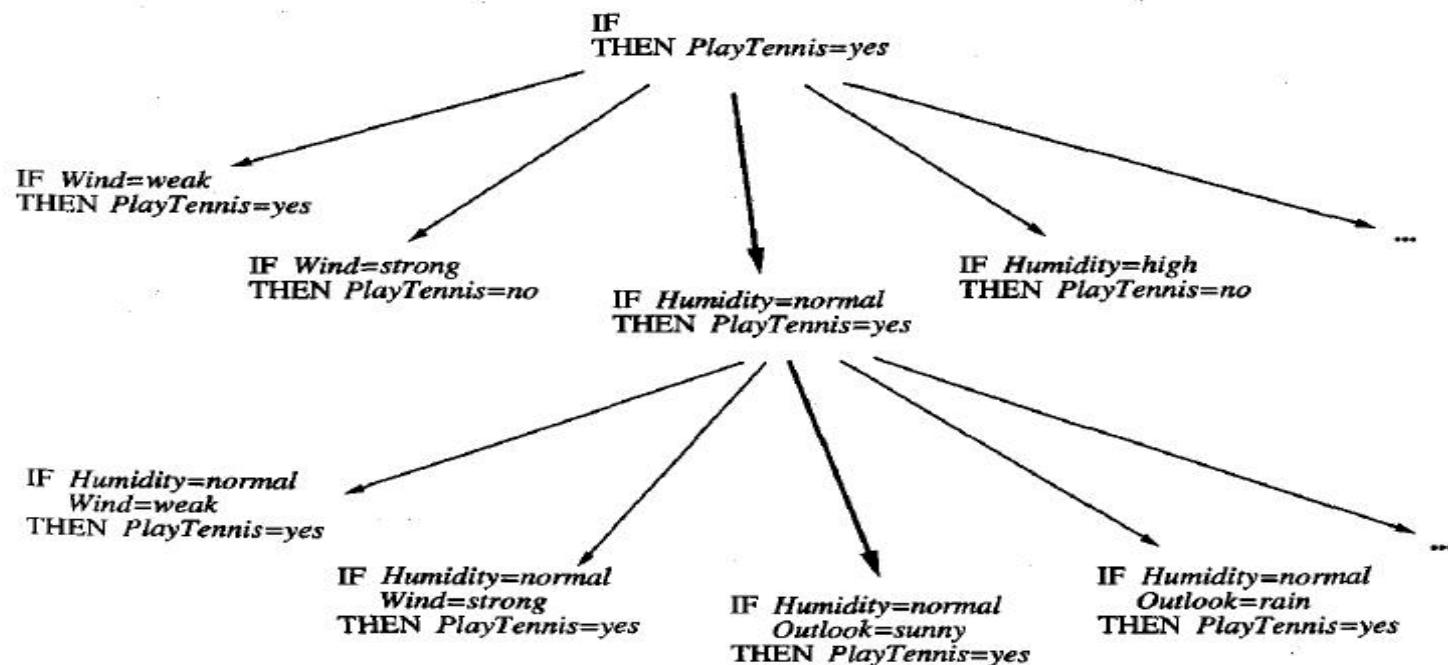


FIGURE 10.1

The search for rule preconditions as LEARN-ONE-RULE proceeds from general to specific. At each step, the preconditions of the best rule are specialized in all possible ways. Rule postconditions are determined by the examples found to satisfy the preconditions. This figure illustrates a beam search of width 1.

LEARN-ONE-RULE(*Target_attribute*, *Attributes*, *Examples*, *k*)

Returns a single rule that covers some of the *Examples*. Conducts a general-to-specific greedy beam search for the best rule, guided by the PERFORMANCE metric.

- Initialize *Best_hypothesis* to the most general hypothesis $\emptyset$
- Initialize *Candidate_hypotheses* to the set $\{Best_hypothesis\}$
- While *Candidate_hypotheses* is not empty, Do
 1. Generate the next more specific candidate_hypotheses
 - *All_constraints* $\leftarrow$ the set of all constraints of the form $(a = v)$, where a is a member of *Attributes*, and v is a value of a that occurs in the current set of *Examples*
 - *New_candidate_hypotheses* $\leftarrow$
 - for each h in *Candidate_hypotheses*,
 - for each c in *All_constraints*,
 - create a specialization of h by adding the constraint c
 - Remove from *New_candidate_hypotheses* any hypotheses that are duplicates, inconsistent, or not maximally specific
 2. Update *Best_hypothesis*
 - For all h in *New_candidate_hypotheses* do
 - If (PERFORMANCE(h , *Examples*, *Target_attribute*)
 - $>$ PERFORMANCE(*Best_hypothesis*, *Examples*, *Target_attribute*))
 - Then *Best_hypothesis* $\leftarrow h$
 3. Update *Candidate_hypotheses*
 - *Candidate_hypotheses* $\leftarrow$ the k best members of *New_candidate_hypotheses*, according to the PERFORMANCE measure.
- Return a rule of the form
 “IF *Best_hypothesis* THEN *prediction*”
 where *prediction* is the most frequent value of *Target_attribute* among those *Examples* that match *Best_hypothesis*.

PERFORMANCE(h , *Examples*, *Target_attribute*)

- $h_examples \leftarrow$ the subset of *Examples* that match h
- return $-Entropy(h_examples)$, where entropy is with respect to *Target_attribute*



Remarks on LEARN-ONE-RULE algorithm

- First, note that each hypothesis considered in the main loop of the algorithm is a **conjunction of attribute-value constraints**.
- Each of these **conjunctive hypotheses corresponds to a candidate set of preconditions** for the rule to be learned and is evaluated by the entropy of the examples it covers
- The **search considers increasingly specific candidate hypotheses** until it reaches a maximally specific hypothesis that contains all available attributes.
- The **postcondition for the output rule** is chosen only in the final step of the algorithm, **after its precondition has been determined**.
- The **algorithm constructs the rule postcondition to predict the value of the target attribute** that is most common among the examples covered by the rule precondition.



2. Variations

Variation 1:

- The **SEQUENTIAL-COVERING** algorithm, together with the **LEARN-ONE-RULE** algorithm, learns a set of if-then rules that covers the training examples.
- **Many variations** on this approach have been explored.
- For example, in some cases **it might be desirable to have the program learn only rules that cover positive examples and to include a "default" that assigns a negative classification** to instances not covered by any rule.
- This approach might be desirable, say, if one is attempting to learn a target concept such as **"pregnant women who are likely to have twins."**
- In this case, the **fraction of positive examples in the entire population is small**, so the rule set will be more compact and intelligible to humans if it identifies only classes of positive examples, with the default classification of all other examples as negative.
- This approach also corresponds to the "negation-as-failure" strategy of PROLOG in, which any expression that cannot be proven to be true is by default assumed to be false



Variation 2:

- Another variation is provided by a **family of algorithms called AQ** (Michalski 1969, Michalski et al. 1986), that predate the CN2 algorithm
- **Like CN2, AQ learns a disjunctive set of rules** that together cover the target function.
- However, **AQ differs in several ways** from the algorithms given here.
 - **First**, the covering algorithm of **AQ differs from the SEQUENTIAL-COVERING** algorithm because it explicitly seeks rules that cover a particular target value, learning a disjunctive set of rules for each target value in turn.
 - **Second, AQ's algorithm for learning a single rule differs from LEARN-ONE-RULE.**
 - While it conducts a general-to-specific beam search for each rule, **it uses a single positive example to focus this search**
 - In particular, **it considers only those attributes satisfied by the positive example** as it searches for progressively more specific hypotheses. Each time it learns a new rule it selects a new positive example from those that are not yet covered



3.LEARNING RULE SETS: SUMMARY

This section considers several **key dimensions** in the design space of such rule learning algorithms.

1.sequential covering algorithms

- First, **sequential covering algorithms** learn one rule at a time, removing the covered examples and repeating the process on the remaining examples

2.LEARN-ONE-RULE

- A second dimension along which approaches vary is the direction of the search in **LEARN-ONE-RULE**. In the algorithm, the **search is from general to specific hypotheses**.
- One advantage** of general to specific search in this case is that there is a single maximally general hypothesis from which to begin the search, whereas there are very many specific hypotheses in most hypothesis spaces



3.generate then test

- A third dimension is whether the LEARN-ONE-RULE search is a **generate then test** search through the syntactically legal hypotheses, as it is in our suggested implementation, or whether it is example-driven so that individual training examples constrain the **generation of hypotheses**

4.post-pruned

- A fourth dimension is whether and how rules are post-pruned. As in decision tree learning, it is possible for LEARN-ONE-RULE to formulate rules that perform very well on the training data, but less well on subsequent data.
- As in decision tree learning, one way to address this issue is to post-prune each rule after it is learned from the training data.
- In particular, **preconditions can be removed from the rule whenever this leads to improved performance over a set of pruning examples distinct from the training examples.**



5. Various evaluation functions have been used and Some common evaluation functions include:

Relative frequency.

- Let n denote the number of examples the rule matches and let n_c denote the number of these that it classifies correctly.
- The relative frequency estimate of rule performance is $\frac{n_c}{n}$

m-estimate of accuracy.

- This accuracy estimate is biased toward the default accuracy expected of the rule.
- let n and n_c denote the number of examples matched and correctly predicted by the rule.
- Let p be the prior probability that a randomly chosen example from the entire data set will have the classification assigned by the rule
- let m be the weight

$$\frac{n_c + mp}{n + m}$$



Entropy.

- This is the measure used by the PERFORMANCE subroutine in the algorithm
- Let S be the set of examples that match the rule preconditions.
- **Entropy** measures the uniformity of the target function values for this set of examples.
- We take the negative of the

$$-Entropy(S) = \sum_{i=1}^c p_i \log_2 p_i$$

have higher scores.

- where c is the number of distinct values the target function may take on, and where p_i is the proportion of examples from S for which the target function takes on the i th value.



A T M E
College of Engineering



LEARNING FIRST-ORDER RULES

- In this section, we consider **learning rules that contain variables-in particular, learning first-order Horn theories.**
- Our motivation for considering such rules is that they are much more expressive than propositional rules.
- **Inductive learning** of first-order rules or theories is often referred to as **inductive logic programming** (or LP for short), because this process can be viewed as automatically inferring PROLOG programs from examples.
- **PROLOG** is a general purpose, Turing-equivalent programming language in which programs are expressed as collections of Horn clauses



A T M E
College of Engineering



1. First-Order Horn Clauses

- To see the **advantages of first-order representations** over propositional (variablefree) representations, consider the task of learning the **simple target concept Daughter (x, y)**, defined over pairs of people x and y.
- The value of **Daughter(x, y)** is **True** when **x is the daughter of y**, and **False otherwise**.
- Suppose **each person in the data is described by the attributes** Name, Mother, Father, Male, Female.
- Hence, each training example will consist of the description of two people in terms of these attributes, along with the value of the target attribute Daughter.



$\langle \text{Name}_1 = \text{Sharon}, \text{Mother}_1 = \text{Louise}, \text{Father}_1 = \text{Bob},$

$\text{Male}_1 = \text{False}, \text{Female}_1 = \text{True},$

• $\text{Name}_2 = \text{Bob}, \text{Mother}_2 = \text{Nora}, \text{Father}_2 = \text{Victor},$

$\text{Male}_2 = \text{True}, \text{Female}_2 = \text{False}, \text{Daughter}_{1,2} = \text{True} \rangle$

he daughter of

IF $(\text{Father}_1 = \text{Bob}) \wedge (\text{Name}_2 = \text{Bob}) \wedge (\text{Female}_1 = \text{True})$

THEN $\text{Daughter}_{1,2} = \text{True}$

Propositional rule

IF $\text{Father}(y, x) \wedge \text{Female}(y),$ THEN $\text{Daughter}(x, y)$

where x and y are variables that can be bound to any person.

first-order representations



- First-order Horn clauses may also refer to variables in the **preconditions that do not occur**
in the postconditions
- IF** $Father(y, z) \wedge Mother(z, x) \wedge Female(y)$
For THEN $GrandDaughter(x, y)$

Note the variable z in this rule, which refers to the father of y , is not present in the rule postconditions



2. Terminology

- Before moving on to algorithms for learning sets of Horn clauses, let us introduce **some basic terminology** from formal logic.
- All expressions are composed of
 - constants (e.g., Bob, Louise),
 - variables (e.g., x, y),
 - predicate symbols (e.g., Married, Greater-Than), and
 - function symbols (e.g., age). The difference between
- predicates and functions is that **predicates take on values of True or False**, whereas **functions may take on any constant as their value**.
- We will use **lowercase symbols for variables** and **capitalized** symbols for constants.
- Also, we will use **lowercase for functions** and **capitalized** symbols for predicates.



- From these symbols, we build up expressions as follows:
 - A term** is any constant, any variable, or any function applied to any term (e.g., Bob, x, age(Bob)).
 - A literal** is any predicate or its negation applied to any term (e.g., Married(Bob, Louise), Greater-Than(age(Sue), 20)).
 - If a **literal contains** a negation ($\neg$) symbol, we call it a **negative literal**, otherwise a **positive literal**.
- A clause** is any **disjunction of literals**, where all variables are assumed to be universally quantified.
- A Horn**

$$H \vee \neg L_1 \vee \dots \vee \neg L_n$$

where H is the positive literal, and $\neg L_1 \dots \neg L_n$ are negative literals. Because of the equalities $(B \vee \neg A) = (B \leftarrow A)$ and $\neg(A \wedge B) = (\neg A \vee \neg B)$, the above Horn clause can alternatively be written in the form

$$H \leftarrow (L_1 \wedge \dots \wedge L_n)$$



which is equivalent to the following, using our earlier rule notation

IF $L_1 \wedge \dots \wedge L_n$, THEN H

Whatever the notation, the Horn clause preconditions $L_1 \wedge \dots \wedge L_n$ are called the clause *body* or, alternatively, the clause *antecedents*. The literal H that forms the postcondition is called the clause *head* or, alternatively, the clause *consequent*.

- Every well-formed expression is composed of *constants* (e.g., *Mary*, 23, or *Joe*), *variables* (e.g., *x*), *predicates* (e.g., *Female*, as in *Female(Mary)*), and *functions* (e.g., *age*, as in *age(Mary)*).
- A *term* is any constant, any variable, or any function applied to any term. Examples include *Mary*, *x*, *age(Mary)*, *age(x)*.
- A *literal* is any predicate (or its negation) applied to any set of terms. Examples include *Female(Mary)*, $\neg \text{Female}(x)$, *Greater_than(age(Mary), 20)*.
- A *ground literal* is a literal that does not contain any variables (e.g., $\neg \text{Female}(\text{Joe})$).
- A *negative literal* is a literal containing a negated predicate (e.g., $\neg \text{Female}(\text{Joe})$).
- A *positive literal* is a literal with no negation sign (e.g., *Female(Mary)*).
- A *clause* is any disjunction of literals $M_1 \vee \dots \vee M_n$ whose variables are universally quantified.
- A *Horn clause* is an expression of the form

$$H \leftarrow (L_1 \wedge \dots \wedge L_n)$$

where $H, L_1 \dots L_n$ are positive literals. H is called the *head* or *consequent* of the Horn clause. The conjunction of literals $L_1 \wedge L_2 \wedge \dots \wedge L_n$ is called the *body* or *antecedents* of the Horn clause.

- For any literals A and B , the expression $(A \leftarrow B)$ is equivalent to $(A \vee \neg B)$, and the expression $\neg(A \wedge B)$ is equivalent to $(\neg A \vee \neg B)$. Therefore, a Horn clause can equivalently be written as the disjunction

$$H \vee \neg L_1 \vee \dots \vee \neg L_n$$

- A *substitution* is any function that replaces variables by terms. For example, the substitution $\{x/3, y/z\}$ replaces the variable x by the term 3 and replaces the variable y by the term z . Given a substitution θ and a literal L we write $L\theta$ to denote the result of applying substitution θ to L .
- A *unifying substitution* for two literals L_1 and L_2 is any substitution θ such that $L_1\theta = L_2\theta$.



LEARNING SETS OF FIRST-ORDER RULES

FOIL(First Order Inductive Learner (FOIL))

- A variety of algorithms has been proposed for learning first-order rules, or Horn clauses.
- In this section we consider a program called FOIL (Quinlan 1990) that employs an approach very similar to the SEQUENTIAL-COVERING and LEARN-ONE-RULE algorithms of the previous section.
- In fact, the FOIL program is the natural extension of these earlier algorithms to first-order representations.
- Formally, the hypotheses learned by FOIL are sets of first-order rules, where each rule is similar to a Horn clause with two exceptions.
 - First, the rules learned by FOIL are more restricted than general Horn clauses, because the literals are not permitted to contain function symbols (this reduces the complexity of the hypothesis space search).
 - Second, FOIL rules are more expressive than Horn clauses, because the literals appearing in the body of the rule may be negated.



A T M E
College of Engineering



- **Applications**
 - It has been demonstrated to learn a recursive definition of the QUICKSORT algorithm
 - It learns to discriminate legal from illegal chess positions.

FOIL(*Target_predicate*, *Predicates*, *Examples*)

- *Pos* $\leftarrow$ those *Examples* for which the *Target_predicate* is *True*
 - *Neg* $\leftarrow$ those *Examples* for which the *Target_predicate* is *False*
 - *Learned_rules* $\leftarrow \{\}$
 - while *Pos*, do
 - Learn a NewRule*
 - *NewRule* $\leftarrow$ the rule that predicts *Target_predicate* with no preconditions
 - *NewRuleNeg* $\leftarrow$ *Neg*
 - while *NewRuleNeg*, do
 - Add a new literal to specialize NewRule*
 - *Candidate_literals* $\leftarrow$ generate candidate new literals for *NewRule*, based on *Predicates*
 - *Best_literal* $\leftarrow \underset{L \in \text{Candidate_literals}}{\text{argmax}} \text{ Foil_Gain}(L, \text{NewRule})$
 - add *Best_literal* to preconditions of *NewRule*
 - *NewRuleNeg* $\leftarrow$ subset of *NewRuleNeg* that satisfies *NewRule* preconditions
 - *Learned_rules* $\leftarrow$ *Learned_rules* + *NewRule*
 - *Pos* $\leftarrow$ *Pos* - {members of *Pos* covered by *NewRule*}
 - Return *Learned_rules*
-

TABLE 10.4

The basic FOIL algorithm. The specific method for generating *Candidate_literals* and the definition of *Foil_Gain* are given in the text. This basic algorithm can be modified slightly to better accommodate noisy data, as described in the text.



- In the FOIL algorithm, the **outer loop corresponds to a variant of the SEQUENTIAL-COVERING algorithm** discussed earlier; that is, it learns new rules one at a time, removing the positive examples covered by the latest rule before attempting to learn the next rule.
- The inner **loop corresponds to a variant of our earlier LEARN-ONE-RULE** algorithm, extended to accommodate first-order rules.
- Note also there are a **few minor differences** between FOIL and these earlier algorithms.
 - In particular, FOIL seeks only rules that **predict when the target literal is True**, whereas our earlier algorithm would seek both rules that predict when it is True and rules that predict when it is False.
 - Also, FOIL performs a **simple hill climbing search** rather than a beam search



A T M E
College of Engineering



- In the outer loop, The **search is a specific-to-general search through** the space of hypotheses, beginning with the most specific empty disjunction and terminating when the hypothesis is sufficiently general to cover all positive training examples.
- The **inner loop of FOIL** performs a finer-grained search to determine the exact definition of each new rule.
- This inner loop searches a second hypothesis space, **consisting of conjunctions of literals**, to find a conjunction that will form the preconditions for the new rule.
- Within this hypothesis space, it conducts a **general-to-specific, hill-climbing search**



A T M E
College of Engineering



The two most substantial differences between FOIL and our earlier SEQUENTIAL-COVERING and LEARN-ONE-RULE algorithm follow from the requirement that it accommodate first-order rules. These differences are:

1. In its **general-to-specific search** to learn each new rule, FOIL employs different detailed steps to generate candidate specializations of the rule. This difference follows from the need to accommodate variables in the rule preconditions.
2. FOIL employs a **PERFORMANCE measure**, Foil-Gain, that differs from the entropy measure



1. Generating Candidate Specializations in FOIL

- To generate candidate specializations of the current rule, **FOIL generates a variety of new literals**, each of which may be individually added to the rule preconditions.
- More precisely: $P(x_1, x_2, \dots, x_k) \leftarrow L_1 \dots L_n$ is
 - where **$L_1 \dots L_n$ are literals forming the current rule preconditions** and where $P(x_1, x_2, \dots, x_k)$ is the literal that forms the rule **head, or postconditions**.
 - FOIL generates candidate specializations of this rule by considering new literals L_{n+1} that fit one of the following forms:
 - $Q(v_1, \dots, v_r)$, where Q is any predicate name occurring in Predicates and where the v_i are either new variables or variables already present in the rule. At least one of the v_i in the created literal must already exist as a variable in the rule.
 - $\text{Equal}(x_j, x_k)$, where x_i and x_k are variables already present in the rule.
 - The negation of either of the above forms of literals.



- To illustrate, **consider learning rules** to predict the target literal **Grand- Daughter(x, y)**, where the other predicates used to describe examples are Father and Female.
- The general-to-specific search in FOIL begins with the most general rule

GrandDaughter(x, y) ←

- **To specialize this initial rule**, the above procedure generates the following literals as candidate additions to the rule preconditions:
 - Equal (x , y) , Female(x), Female(y), Father(x, y), Father(y, x), Father(x, z), Father(z, x), Father(y, z), Father(z, y), and the negations of each of these literals (e.g., $\neg$ Equal(x, y)).
 - Note that z is a new-variable here, whereas x and y exist already within the current rule.
- Now suppose that among the above literals FOIL greedily selects **Father(y, z)** as the most promising, leading to the more specific rule

GrandDaughter(x, y) ← Father(y, z)



- In generating candidate literals to **further specialize this rule**, FOIL will now **consider all of the literals mentioned in the previous step, plus the additional literals**
 - Female(z), Equal(z, x), Equal(z, y), Father(z, w), Father(w, z), and their negations.
 - These new literals are considered at this point because the variable z was added to the rule in the previous step.
 - Because of this, FOIL now considers an additional **new variable w**.
- If FOIL at this point were to **select the literal Father(z, x)** and on the next iteration select the literal **Female(y)**, this would lead to the following rule, which covers only positive examples and hence terminates the search for **further specializations of the rule**.

$$GrandDaughter(x, y) \leftarrow Father(y, z) \wedge Father(z, x) \wedge Female(y)$$

- At this point, FOIL will remove all positive examples covered by this new rule



2.Guiding the Search in FOIL

- To select the most promising literal from the candidates generated at each step, **FOIL considers the performance of the rule over the training data.**
- In doing this, **it considers all possible bindings of each variable in the current rule.**
- To illustrate this process, consider again the example in which we seek to learn a set of rules for the target literal **GrandDaughter(x, y).**
- For illustration, assume the training data includes the following simple set of assertions, where we use the convention that **P(x, y) can be read as "The P of x is y."**
 - GrandDaughter(Victor, Sharon)
 - Father(Sharon, Bob)
 - Father(Tom, Bob)
 - Female(Sharon)
 - Father(Bob, Victor)



- To select the best specialization of the current rule, FOIL considers each distinct way in which the rule variables can bind to constants in the training data. $GrandDaughter(x, y) \leftarrow$
- For example, in the initial step when the rule is $GrandDaughter(x, y) \leftarrow$
- Given the four possible constants, there are 16 possible variable bindings for this initial rule.
- We will use the notation $\{x/Bob, y/Sharon\}$ to denote a particular variable binding; that is, a substitution mapping each variable to a constant.
- The binding $\{x/victor, y/Sharon\}$ corresponds to a positive example binding, because the training data includes the assertion $GrandDaughter(Victor, Sharon)$.
- The other 15 bindings allowed by the rule (e.g., the binding $\{x/Bob, y/Tom\}$) constitute negative evidence for the rule in the current example,
- Note if a literal is added that introduces a new variable, then the bindings for the rule will grow in length (e.g., if $Father(y, z)$ is added to the above rule, then the original binding $\{x/victor, y/Sharon\}$ will become the more lengthy $\{x/victor, y/Sharon, z/Bob\}$.



- The **evaluation function** used by **FOIL** to **estimate the utility of adding a new literal** is based on the **numbers of positive and negative bindings covered** before and after adding the new literal.
- More precisely, consider some **rule R**, and a candidate literal **L** that might be added to the body of **R**.
- Let **R'** be the rule created by adding literal **L** to rule **R**.
- The value **Foil-Gain(L, R)** of adding **L** to **R** is defined as

$$\text{Foil_Gain}(L, R) \equiv t \left(\log_2 \frac{p_1}{p_1 + n_1} - \log_2 \frac{p_0}{p_0 + n_0} \right)$$

where **p₀** is the number of positive bindings of rule **R**, and **n₀** is the number of negative bindings of **R**, and **p₁** and **n₁** are the corresponding numbers for rule **R'**.

p₁ is the number of positive bindings of rule **R'**, and **n₁** is the number of negative bindings of **R'**. Finally, **t** is the number of positive bindings of rule **R** that are still covered after adding literal **L** to **R**.



3. Learning Recursive Rule Sets

- if we include the target predicate in the input list of Predicates, then FOIL will consider it as well when generating candidate literals.
- This will allow it to form **recursive rules**-rules that use the same predicate in the body and the head of the rule.
- For
relat

IF	$Parent(x, y)$	THEN	$Ancestor(x, y)$	f the Ancestor
IF	$Parent(x, z) \wedge Ancestor(z, y)$	THEN	$Ancestor(x, y)$	